**Prep Notes**

1. Increase text size in Visual Studio
2. Turn off whitespace indicators
3. Familiarize with zoom tool
4. Set Client, VulnServer1, AttackerVisual as startup projects
5. Have a command window open in the Attacker1/bin/debug folder (with previous results cleared)
   1. .\Attacker1 forge 12345689 TEST 8.88

**Intro:**

1. Two quick slides to set up the scenario and then we’ll jump into some code and demos
2. I’m not a crypto expert, but I enjoy digging into this kind of thing
3. I continue to learn and I welcome any corrections on this material

**Coding:**

1. Utils…QuickCrypto.cs: Paste encrypt/decrypt methods from MSDN.
   1. *Snippet.txt in case internet is not up.*
   2. Fix the comment that wraps to two lines
   3. Make the methods public
2. Client…Form1.cs: complete the code behind the Generate button
   1. Snippet.txt if you forget
   2. Hmm, where do we get the IV? Refer back to the MSDN. Not clear how we should handle it.
   3. Let’s assume it’s like the encryption key and doesn’t change every time
3. Run it
   1. Looks nicely random and unreadable
      1. Size is double since I’m using two hex characters for each byte of the ciphertext. Could have used base64 but let’s keep it simple.
   2. But wait… notice that when I regenerate the token, only the last block changes
   3. AES is a block cipher, which means it operates on a whole block at a time (16 bytes – 128 bits in this case).
      1. So you notice that any tiny change in the timestamp portion of the plain text makes that whole block change in unpredictable ways.
      2. But it does nothing to scramble the earlier blocks that didn’t change. This means our goal of secrecy is a failure, because you can tell if the account numbers are the same on two blocks just by looking at the cipher text.
      3. Note that a change in the account number causes changes to cascade to the later blocks too. We’ll come back to this later.
4. Fix to use QuickCrypto.GetRandomIV
   1. The IV will be required to decrypt at other end, but it’s not a secret so we’ll just include it as the first block of ciphertext (**don’t forget to add this line**)
   2. Now the whole thing changes every time even if the plaintext didn’t change
5. Let’s take a step back and look at how padding works in a block cipher. Since we need blocks of 16 bytes, the algorithm has to pad out the remainder of the final block according to some algorithm.
   1. The most common standard (and the default in .Net) is PKCS#7, which basically says you pad out those final bytes of the plaintext with the numeric value of how many bytes of padding.
   2. Show the decrypt button with the in the WinForms app
6. Show the server API that accepts the ticket
   1. <http://localhost:55165/api/transaction?ticket=blah>
   2. Note that if we change something in the middle of the ticket we get a validation error status, but if we change something near the end we get a different status.
   3. Show the server-side code and explain that basically the only reason the exception is thrown is because of bad padding
7. Back to slide deck for CBC mode (the default in .Net’s AES algorithm)
8. Run the padding oracle attack (command-line app)
   1. Paste one of the tickets into
9. You might think, “well, that requires you to intercept one of the tokens, but you couldn’t forget one from scratch”
10. But look at the diagram again: what we’ve done in the first part here is use the padding oracle to give us a decryption oracle
11. Run the CBC-R forge utility from the Attacker1/bin/debug
12. Attacker1.exe forge <desired token value>